Virtual Functions and Polymorphism Solutions

Copyright © James Raynard 2021

Static and dynamic types

- Explain what is meant by static and dynamic binding when calling a member function on an object
  - Static and dynamic binding refer to the two different ways in which a member function can be called on an object
  - In static binding (the norm), the declared type of the object is used
  - In dynamic binding, the type of the object in memory is used
- What two conditions must apply for dynamic binding to occur?
  - The member function is called on a pointer or reference to a base class
  - The member function is declared virtual in the base class

Copyright © James Raynard 2021

Virtual Member Functions

- What is a virtual member function?
  - A virtual member function has the virtual keyword before it
  - When a virtual member function is called on a pointer or reference to a base class, it will be called on the dynamic type of the instance

```
Circle circle;
Drawable& dref = circle;
dref.draw();            // Calls Circle's draw(), not Drawable's
```

Copyright © James Raynard 2021

Function call arguments

- Using the code on the next page, write a program which creates a Circle object and passes it to draw_shape()
- Make the draw() member function in Drawable virtual and run your program again
- Add a Triangle class, which is also derived from Drawable. Alter your program so that it creates a Triangle object and passes it to draw_shape()
- Explain your observations

Copyright © James Raynard 2021

Function call arguments

- In the first case, draw() is non-virtual. This means the static type will always be used for dref, even though it is a reference to the base class. The base class's draw() member function will always be called
- In the second case, draw() is virtual. This means the dynamic type will always be used for dref, since it is a reference to the base class. Circle's draw() member function will be called
- The Triangle argument is derived from Drawable, so its dynamic type is used and Triangle's draw() member function is called. Notice that we did not need to write any extra code to make this new class work correctly with draw_shape()

Copyright © James Raynard 2021

Function call arguments code

```
class Drawable {
        public:
        void draw() const { cout << "What am I supposed to do here?\n"; }
};

class Circle : public Drawable {
        public:
        void draw() const { cout << "I'm a circle!\n"; }
};

void draw_shape(const Drawable& dref) {
   dref.draw();
}
```

Copyright © James Raynard 2021

Polymorphism

- Explain briefly what is meant by "polymorphism"
  - Polymorphism means that different types have the same interface
- How do the programs in this video demonstrate polymorphism?
  - All the classes in the Drawable hierarchy have the same interface, so in effect we can use them interchangeably

Copyright © James Raynard 2021

Pure Virtual Member Functions

- What is meant by a "pure virtual member function"?
  - A pure virtual member function has "=0" instead of a function body
  - It is undefined in the class, but can be overridden in derived classes
- What effect does making a member function pure virtual have?
  - If a class has a pure virtual member function, we cannot create objects of it
  - Every child of this class must implement the member function
- In the program you wrote in the previous exercise, change Drawable::draw() to be a pure virtual function
- Compile and run your program again. What happens if you try to create a Drawable object?
  - Compiler error - cannot create a variable of abstract type

Copyright © James Raynard 2021

Abstract Base Class

- What is an "abstract base class"?
  - An abstract base class contains a pure virtual member function
- Why is it useful?
  - It expresses clearly that the base class is an interface to the class hierarchy, rather than a concrete class which represents some entity in the program
  - If we accidentally use a value object for the base class instead of a reference or pointer, we get a compiler error instead of silently having our objects "sliced"

Copyright © James Raynard 2021

Virtual destructor in base class

- Explain why a base class should always have a virtual destructor implemented
  - The default destructor generated by the compiler is non-virtual
  - This means that static binding is always used for the destructor
  - If called on a pointer or reference which has a different dynamic type, this means that the derived class part of the instance is not destroyed, resulting in a partially destroyed object
  - Providing a virtual destructor will ensure that dynamic binding is always used

Copyright © James Raynard 2021